# 为什么使用泛型

简而言之，泛型在定义类，接口，和方法时使类型变得参数化。这很像在方法声明中使用的形式参数，参数化类型为我们提供了在不同的输入参数下重用代码的方式。不同的是，形式参数的输入是值，而类型参数的输入是类型。

使用泛型的代码比非泛型代码有很多好处：

* 编译时期更加强大的类型检查

Java编译器对泛型代码提供强类型检查，如果代码违反了类型安全，编译器将报错。修复编译时期错误要比运行时期错误更加容易，因为编译时期错误很难发现。

* 消除类型转换

不使用泛型的代码片段

List list = new ArrayList();

list.add("hello");

String s = (String) list.get(0);

当使用泛型重写时，代码不需要进行类型转换

List<String> list = new ArrayList<String>();

list.add("hello");

String s = list.get(0); // no cast

* 使程序员能够实现通用算法。

通过使用泛型，程序员可以实现对不同类型的集合进行操作的通用算法，可以自定义，并且类型安全易于阅读

# 泛型类型

一个泛型类型是一个对类型进行参数化的泛型类或者接口。下面将通过修改以下Box类来演示该概念。

## 一个简单的Box类

首先检查对任意类型的对象进行操作的非泛型Box类。它只需要提供两个方法:向box中添加对象的set方法，以及检索对象的get方法。

public class Box {

private Object object;

public void set(Object object) { this.object = object; }

public Object get() { return object; }

}

由于它的方法接受或返回一个Object，所以你可以随意传入任何你想要的传入的，只要它不是一个原始类型。这些方法在编译时期无法验证类的使用方式。在代码的某一部分可能在box中放入了一个Integer，并期望从其中获取Integer，而另一部分代码中可能会错误地传入一个String，从而导致运行时错误。

## 泛型版本的Box类

泛型类使用以下格式定义：

class name<T1, T2, ..., Tn> { /\* ... \*/ }

类型参数部分由尖括号（<>）分隔，跟在类名之后。它指定类型参数（也称为类型变量）T1，T2，...和Tn。

要更新Box类以使用泛型，您可以通过将代码“public class Box”更改为“public class Box <T>”来创建泛型类型声明。这引入了可以在类中任何地方使用的类型变量T。

/\*\*

\* Generic version of the Box class.

\* @param <T> the type of the value being boxed

\*/

public class Box<T> {

// T stands for "Type"

private T t;

public void set(T t) { this.t = t; }

public T get() { return t; }

}

您可以看到，所有出现的Object都被替换为T。类型变量可以是您指定的任何非基本类型：任何类类型，任何接口类型，任何数组类型，或甚至其他类型变量。

## 类型参数命名约定

按照惯例，类型参数名称是单个大写字母。这与您已经知道的变量命名惯例形成鲜明对照，并且有很好的理由：没有这个约定，很难说出类型变量和普通类或接口名称之间的区别。

最常用的类型参数名称是：

* E——元素（Java Collections Framework广泛使用）
* K——键（key）
* N——数字（Number）
* T——类型（Type）
* V——值（Value）
* S,U,V——第二，三，四类型

您将看到这些名称在整个Java SE API和本课程的其余部分使用

## 调用和实例化泛型类型

要从你的代码中引用泛型的Box类，你必须执行一个泛型类型的调用，用一些具体的值替换T，比如Integer：

Box<Integer> integerBox;

您可以将泛型类型调用视为与普通方法调用相似，但不是将参数传递给方法，而是将类型参数（在这种情况下为Integer）传递给Box类本身。

Type Parameter 和 Type Argument

许多开发人员可以互换使用术语Type Parameter和Type Argument，但是这些术语是不一样的。当编写代码时，为了创建要给参数化类型，提供了一个type argument。因此，T在Foo<T>中是一个type parameter而String在Foo<String>中是一个type argument。使用这些术语时，本课程将遵守此定义。

像任何其他变量声明一样，此代码实际上并没有创建一个新的Box对象。它只是声明integerBox会保存一个“Box of Ingteger“的引用，该引用指明如何读取Box <Integer>。

泛型类型的调用通常称为参数化类型。

要实例化此类，请像往常一样使用new关键字，但在类名和括号之间放置<Integer>：

Box<Integer> integerBox = new Box<Integer>();

## 菱形符号（The Diamond）

在Java SE 7及更高版本中，只要编译器可以从上下文中确定或推断类型参数，就可以使用一组空的类型参数（<>）替换调用泛型构造函数所需的类型参数。例如，您可以使用以下语句创建Box <Integer>的实例：

Box<Integer> integerBox = new Box<>();

有关菱形符号和类型推断的更多信息，请参阅类型推断。

## 多类型参数

如前所述，泛型类可以有多个类型参数。例如，泛型的OrderedPair类，它实现了泛型的Pair接口：

public interface Pair<K, V> {

public K getKey();

public V getValue();

}

public class OrderedPair<K, V> implements Pair<K, V> {

private K key;

private V value;

public OrderedPair(K key, V value) {

this.key = key;

this.value = value;

}

public K getKey() { return key; }

public V getValue() { return value; }

}

以下语句创建了OrderedPair类的两个实例化：

Pair<String,Integer> p1 = newOrderedPair<String, Integer>("Even", 8);

Pair<String,String> p2 = new OrderedPair<String, String>("hello", "world");

代码中，new OrderedPair<String,Integer>实例化K为一个String,V为一个Integer。因此OrderedPair的构造方法的类型参数分别为String和Integer。由于自动装箱，将String和int传递给该类也是有效的。

如The Diamond所述，由于Java编译器可以从声明OrderedPair <String，Integer>推断出K和V类型，所以可以使用菱形符号来缩短这些语句：

OrderedPair<String, Integer> p1 = new OrderedPair<>("Even", 8);

OrderedPair<String, String> p2 = new OrderedPair<>("hello", "world");

要创建泛型接口，请遵循与创建泛型类相同的规定。

## 参数化类型

您也可以使用参数化类型（即List <String>）来作为类型参数（即K或V）。例如，使用OrderedPair <K，V>示例：

OrderedPair<String, Box<Integer>> p = new OrderedPair<>("primes", new Box<Integer>(...));

# 原生类型（raw type）

原生类型（raw type）是没有任何类型参数的泛型类或接口的名称。例如，给定泛型Box类：

public class Box<T> {

public void set(T t) { /\* ... \*/ }

// ...

}

要创建一个参数化类型的Box <T>，您可以为形式类型参数T提供一个实际的类型参数：

Box<Integer> intBox = new Box<>();

如果实际的类型参数被省略，您将创建一个Box <T>的原生类型：

Box rawBox = new Box();

因此，Box是泛型类型Box <T>的原生类型。但是，非泛型类或接口类型不是原生类型。

原生类型存在于许多遗留代码中，因为许多API类（如Collections类）在JDK 5.0之前不是泛型的。当使用原生类型时，您实质上是获得了一个预泛型 - 一个提供Object的Box。为了向后兼容，允许将参数化类型分配给其原生类型：

Box<String> stringBox = new Box<>();

Box rawBox = stringBox; // OK

但是，如果将原始类型分配给参数化类型，则会收到警告：

Box rawBox = new Box(); // rawBox is a raw type of Box<T>

Box<Integer> intBox = rawBox; // warning: unchecked conversion

如果您使用原生类型来调用在相应的泛型类型中定义的泛型方法，则还会收到警告：

Box<String> stringBox = new Box<>();

Box rawBox = stringBox;

rawBox.set(8); // warning: unchecked invocation to set(T)

警告显示，原生类型绕过泛型类型检查，将不安全代码的捕获推迟到运行时。因此，您应避免使用原生类型。

在类型擦除章节有关于Java编译器如何使用原生类型的更多信息。

## 未检查错误信息

如前所述，当将旧代码与泛型代码混用时，您可能会遇到类似于以下内容的警告消息：

Note: Example.java uses unchecked or unsafe operations.

Note: Recompile with -Xlint:unchecked for details.

当使用旧的API对一个原生类型进行操作的时候，可能会发生这种情况，如以下示例所示：

public class WarningDemo {

public static void main(String[] args){

Box<Integer> bi;

bi = createBox();

}

static Box createBox(){

return new Box();

}

}

“unchecked”表示编译器没有足够的类型信息来执行所有必要的类型检查以确保类型安全。尽管编译器提供了一个提示，但是默认情况下，“unchecked”警告是被禁用的。要查看所有“unchecked”警告，请使用Xlint:unchecked重新编译。

使用-Xlint：unchecked重新编译前面的示例可以显示以下附加信息：

WarningDemo.java:4: warning: [unchecked] unchecked conversion

found : Box

required: Box<java.lang.Integer>

bi = createBox();

^

1 warning

要完全禁用“unchecked”警告，请使用-Xlint：unchecked标志。@SuppressWarnings（“unchecked”）注释也会抑制“unchecked”的警告。如果您不熟悉@SuppressWarnings语法，请参阅注释。

# 泛型方法

泛型方法是引入自己的类型参数的方法。这类似于声明泛型类型，但类型参数的范围仅限于声明它的方法。允许静态和非静态泛型方法，以及泛型类构造函数。

泛型方法的语法包括一个类型参数，位于尖括号内，并出现在方法的返回类型之前。对于静态泛型方法，类型参数部分必须出现在方法的返回类型之前。

Util类包括一个泛型的方法compare，用来比较两个Pair对象：

public class Util {

public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {

return p1.getKey().equals(p2.getKey()) &&

p1.getValue().equals(p2.getValue());

}

}

public class Pair<K, V> {

private K key;

private V value;

public Pair(K key, V value) {

this.key = key;

this.value = value;

}

public void setKey(K key) { this.key = key; }

public void setValue(V value) { this.value = value; }

public K getKey() { return key; }

public V getValue() { return value; }

}

调用此方法的完整语法是：

Pair<Integer, String> p1 = new Pair<>(1, "apple");

Pair<Integer, String> p2 = new Pair<>(2, "pear");

boolean same = Util.<Integer, String>compare(p1, p2);

上面泛型方法显示给出了类型参数，但一般来说，这可以省略，编译器会推断需要的类型：

Pair<Integer, String> p1 = new Pair<>(1, "apple");

Pair<Integer, String> p2 = new Pair<>(2, "pear");

boolean same = Util.compare(p1, p2);

此功能称为类型推断（type inference），允许您以常规方法调用泛型方法，而不在尖括号内指定类型。

# 有界类型参数

可能有时候要限制在参数化类型中可以用作类型参数的类型。例如，一个对数字进行操作的方法可能只希望接受Number或其子类的实例。这就是有界类型参数。

要声明一个有界类型的参数，需要列出类型参数的名称，后跟extends关键字，再后跟其上边界，在此示例中为Number。请注意，在这种情况下，extends在一般意义上用于表示“extends”（如在类中）或“implements”（如在接口中）。

public class Box<T> {

private T t;

public void set(T t) {

this.t = t;

}

public T get() {

return t;

}

public <U extends Number> void inspect(U u){

System.out.println("T: " + t.getClass().getName());

System.out.println("U: " + u.getClass().getName());

}

public static void main(String[] args) {

Box<Integer> integerBox = new Box<Integer>();

integerBox.set(new Integer(10));

integerBox.inspect("some text"); // error: this is still String!

}

}

在修改我们的泛型方法，添加了这个有界类型的参数后，编译现在将失败，因为我们在调用inspect时传入了一个String：

Box.java:21: <U>inspect(U) in Box<java.lang.Integer> cannot

be applied to (java.lang.String)

integerBox.inspect("10");

^

1 error

除了限制可用于实例化通用类型的类型之外，有界类型参数还允许您调用边界对象中定义的方法：

public class NaturalNumber<T extends Integer> {

private T n;

public NaturalNumber(T n) { this.n = n; }

public boolean isEven() {

return n.intValue() % 2 == 0;

}

// ...

}

isEven方法通过n调用Integer类中定义的intValue方法。

## 多重边界

前面示例中说明了在单边界类型参数的使用，但类型参数可以有多个边界：

<T extends B1 & B2 & B3>

一个有多个边界的类型变量是在边界中所有类型的子类型。如果其中一个边界是一个类，则必须首先指定。例如：

Class A { /\* ... \*/ }

interface B { /\* ... \*/ }

interface C { /\* ... \*/ }

class D <T extends A & B & C> { /\* ... \*/ }

如果指定边界A为第一个，那么就会产生一个编译时期错误:

class D <T extends B & A & C> { /\* ... \*/ } // compile-time error

# 泛型，继承和子类型

如您所知，如果类型是兼容的，则可以将一种类型的对象分配给另一种类型的对象。例如，您可以分配一个Integer给一个Object，因为Object是Integer的父类型之一：

Object someObject = new Object();

Integer someInteger = new Integer(10);

someObject = someInteger; // OK

在面向对象的术语中，这被称为“是一个（is a）”关系。因为Integer是一种Object，所以这种赋值是允许的。但是Integer同时也是一种Number,所以下面的代码也是有效的：

public void someMethod(Number n) { /\* ... \*/ }

someMethod(new Integer(10)); // OK

someMethod(new Double(10.1)); // OK

泛型同样如此。您可以执行一个泛型类型调用，将Number作为其类型参数传递，并且如果参数与Number兼容，则将允许任何后续的add方法调用：

Box<Number> box = new Box<Number>();

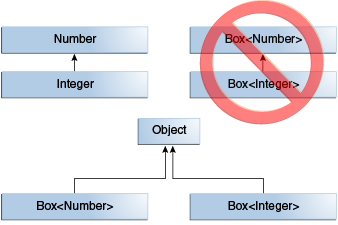
box.add(new Integer(10)); // OK

box.add(new Double(10.1)); // OK

现在考虑以下方法：

public void boxTest(Box<Number> n) { /\* ... \*/ }

它接受什么类型的论据？通过查看其签名，您可以看到它接受一个类型为Box <Number>的单个参数。但是，这是什么意思？它是否允许按照您的预期传递Box <Integer>或Box <Double>？答案是“否”，因为Box <Integer>和Box <Double>不是Box <Number>的子类型。

这是在使用泛型编程时一个最常见的误解，但这是一个重要的学习概念

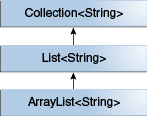
即使Integer是Number的子类型，Box <Integer>不是Box <Number>的子类型。

注意：给定两个具体类型A和B（例如Number和Integer），MyClass <A>与MyClass <B>无关，无论A和B是否相关。 MyClass <A>和MyClass <B>的父对象是Object。

## 泛型类和子类型

可以通过继承或实现一个泛型类或接口，来创建一个它的子类型。一个类或接口的类型参数与另一个的类型参数之间的关系，是由extends和implement来决定。

以Collections类作为示例，ArrayList <E>实现List <E>，并且List <E>扩展Collection <E>。所以ArrayList <String>是List <String>的子类型，它是Collection <String>的子类型。只要不改变类型参数，子类型之间的关系将保留在类型之间。



现在想象我们要定义我们自己的List接口PayloadList，它将泛型类型P的可选值与每个元素相关联。其声明可能如下所示：

interface PayloadList<E,P> extends List<E> {

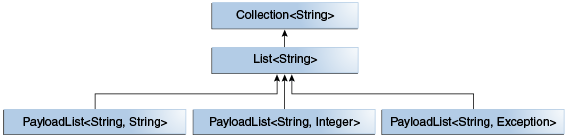
void setPayload(int index, P val);

...

}

以下PayloadList参数都是List <String>的子类型：

* PayloadList<String,String>
* PayloadList<String,Integer>
* PayloadList<String,Exception>



# 类型推断

类型推断是Java编译器查看每个方法调用和相应声明的能力，以确定调用适用的类型参数（或参数）。推断算法确定参数的类型，如果可用的话，这个类型将被分配或返回。最后，推断算法尝试找到适用于所有参数的最适应类型。

为了说明这个最后一点，在下面的例子中，推断确定传递给pick方法的第二个参数的类型是Serializable：

static <T> T pick(T a1, T a2) { return a2; }

Serializable s = pick("d", new ArrayList<String>());

在这种情况下，我们使用两种不同类型的T：String和ArryaList。编译器然后使用最普适的类型参数，使得方法调用类型正确。在这种情况下，它推断T是两种类型共同实现接口类型Serializable。

## 类型推断和泛型方法

在泛型方法中引入了类型推断，这使您能够像普通的方法一样调用泛型方法，而不在尖括号之间指定类型，考虑下面的例子BoxDemo，它需要Box类：

public class BoxDemo {

public static <U> void addBox(U u,

java.util.List<Box<U>> boxes) {

Box<U> box = new Box<>();

box.set(u);

boxes.add(box);

}

public static <U> void outputBoxes(java.util.List<Box<U>> boxes) {

int counter = 0;

for (Box<U> box: boxes) {

U boxContents = box.get();

System.out.println("Box #" + counter + " contains [" +

boxContents.toString() + "]");

counter++;

}

}

public static void main(String[] args) {

java.util.ArrayList<Box<Integer>> listOfIntegerBoxes =

new java.util.ArrayList<>();

BoxDemo.<Integer>addBox(Integer.valueOf(10), listOfIntegerBoxes);

BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes);

BoxDemo.addBox(Integer.valueOf(30), listOfIntegerBoxes);

BoxDemo.outputBoxes(listOfIntegerBoxes);

}

}

以下是此示例的输出：

Box #0 contains [10]

Box #1 contains [20]

Box #2 contains [30]

泛型方法addBox定义了一个名为U的类型参数。通常，Java编译器可以推断泛型方法调用的类型参数。因此，在大多数情况下，您不必指定它们。例如，要调用泛型方法addBox，可以使用类型见证指定类型参数，如下所示：

BoxDemo.<Integer>addBox(Integer.valueOf(10), listOfIntegerBoxes);

或者，如果省略类型见证，Java编译器会自动推断（从方法的参数），类型参数是Integer：

BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes);

## 类型推断和泛型类

只要编译器可以从上下文中推断出类型参数，就可以使用空的一组类型参数（<>）替换调用泛型构造函数所需的类型参数。这对尖括号是非正式地称为菱形。

例如，考虑以下变量声明：

Map<String, List<String>> myMap = new HashMap<String, List<String>>();

您可以使用空的一组类型参数（<>）替换构造函数的参数化类型：

Map<String, List<String>> myMap = new HashMap<>();

请注意，要在泛型类实例化过程中使用类型推断，您必须使用菱形符号。在以下示例中，编译器生成未经检查的转换警告，因为HashMap（）构造函数引用了HashMap原生类型，而不是Map <String，List <String >>类型：

Map<String, List<String>> myMap = new HashMap(); // unchecked conversion warning

## 类型推断和泛型类以及非泛型类的泛型构造方法

请注意，在泛型类以及非泛型类中，构造方法都可以是泛型（换而言之，声明自己的形式类型参数）。考虑以下示例

class MyClass<X> {

<T> MyClass(T t) {

// ...

}

}

考虑下面的类MyClass的实例化：

new MyClass<Integer>("")

此语句创建一个参数化类型MyClass <Integer>的实例; 该语句显式指定了泛型类MyClass <X>的形式类型参数X的类型为Integer。请注意，此泛型类的构造函数包含一个形式类型参数T. 编译器推断出此泛型类的构造函数的形式类型参数T的类型为String（因为此构造函数的实际参数是String对象）。

在Java SE 7之前的版本的编译器能够推断泛型构造函数的实际类型参数，类似于泛型方法。但是，J在ava SE 7及更高版本中如果您使用菱形符号（<>），编译器可以推断要实例化的泛型类的实际类型参数。请考虑以下示例：

MyClass<Integer> myObject = new MyClass<>("");

在这个例子中，编译器推断泛型类Myclass<x>中形式参数X的类型为Integer。它也推断出泛型类的构造方法的形式参数T额类型为String。

需要注意的是，推断算法只使用调用参数，目标类型以及可能的明显的预期返回类型来推断类型。推断算法不使用程序后期的结果。

## 目标类型

Java编译器利用目标类型来推断泛型方法调用的类型参数。表达式的目标类型是Java编译器期望的数据类型，具体取决于表达式的出现位置。考虑方法Collections.emptyList，其声明如下：

static <T> List<T> emptyList();

考虑以下赋值语句：

List<String> listOne = Collections.emptyList();

此语句期待List <String>的实例;这个数据类型就是是目标类型。因为方法emptyList返回一个类型为List <T>的值，编译器会推断类型参数T一定是值String。这可以在Java SE 7和8中使用。或者，您可以使用类型见证并指定T的值如下：

List<String> listOne = Collections.<String>emptyList();

但是，在这种情况下这不是必需的。但在其他情况下是有必要的。请考虑以下方法：

void processStringList(List<String> stringList) {

// process stringList

}

假设你想使用一个空List调用processStringList方法。在Java SE 7中，以下语句不能编译：

processStringList(Collections.emptyList());

Java SE 7编译器会生成类似于以下内容的错误消息：

List<Object> cannot be converted to List<String>

编译器需要一个类型参数T的值，所以它从值Object开始。因此，对Collections.emptyList的调用返回一个类型为List <Object>的值，它与processStringList方法不兼容。因此，在Java SE 7中，必须指定type参数的值的值如下：

processStringList(Collections.<String>emptyList());

# 通配符

在泛型代码中，问号（？）被称为通配符，表示一个未知类型。通配符可以用在多种情景下：作为参数的类型，字段，或者局部变量；有时也作为返回类型（尽管具体指定类型会是更好的编程实践）。通配符从不被用作泛型方法调用类型参数，泛型类实例创建或超类型。

## 上边界通配符

您可以使用上边界通配符来放宽对变量的限制。例如，假设你想编写一个适用于List <Integer>，List <Double>和List <Number>的方法;您可以通过使用上限通配符来实现此目的。

要声明一个上限通配符，请使用通配符（'？'），后跟extends关键字，再后跟其上边界。请注意，在这种情况下，extends在一般意义上用于表示“extends”（如在类中）或“implements”（如在接口中）。

要写一个适用于Number以及它的子类型，比如Integer，Double和Float的list的方法，应当指定 List<? extends Number>。List <Number>一词比List <？extends Number>限制更大，因为前者匹配一个类型为Number的list，而后者匹配一个类型为Number或其任何子类的list。

考虑下面的process方法

public static void process(List<? extends Foo> list) { /\* ... \*/ }

上边界通配符<? extends Foo>中，其中Foo是任何类型，匹配Foo和任何Foo的子类。Process方法可以以Foo类型来访问集合元素

public static void process(List<? extends Foo> list) {

for (Foo elem : list) {

// ...

}

}

在foreach子句中，elem变量遍历列表中的每个元素。 Foo类中定义的任何方法现在可以在elem上使用。

sumOfList方法返回列表中的数字之和：

public static double sumOfList(List<? extends Number> list) {

double s = 0.0;

for (Number n : list)

s += n.doubleValue();

return s;

}

## 无边界通配符

无边界通配符类型是通过通配符（？）来指定。比如List<？>。这个被称为未知类型list。只有在以下两种情况下，无界通配符是才是有用的：

* 如果您正在编写可以使用Object类中提供的功能实现的方法。
* 当代码使用泛型类中不依赖类型参数的方法时。比如List.size或者List.clear。事实上，Class <？>是经常使用的，因为Class <T>中的大多数方法都不依赖于T.

考虑下面的方法printList

public static void printList(List<Object> list) {

for (Object elem : list)

System.out.println(elem + " ");

System.out.println();

}

printList的目标是打印任何类型的list，但是它无法实现该目的 - 它仅打印一个Object实例list;它不能打印List <Integer>，List <String>，List <Double>等，因为它们不是List <Object>的子类型。要编写一个泛型的printList方法，请使用List <？>：

public static void printList(List<?> list) {

for (Object elem: list)

System.out.print(elem + " ");

System.out.println();

}

因为对于任何具体类型A，List <A>是List <？>的子类型，您可以使用printList打印任何类型的list：

List<Integer> li = Arrays.asList(1, 2, 3);

List<String> ls = Arrays.asList("one", "two", "three");

printList(li);

printList(ls);

本课程中的示例中使用了Arrays.asList方法。这个静态工厂方法转换指定的数组并返回一个固定大小的list。

请注意，List <Object>和List <？>不一样。您可以将Object或Object的任何子类型插入到List <Object>中。但是您只能将null插入List<？>。

## 下边界通配符

下边界通配符将未知类型限制为该类型的指定类型或父类型。下限通配符用通配符（'？'）表示，后跟super关键字，后跟其下界：<？super A>。

您可以指定通配符的上限，也可以指定下限，但不能同时指定

假设你想编写一个将Integer对象放入list的方法。为了最大化代码的灵活性，您希望该方法适用于List <Integer>，List <Number>和List <Object> - 任何可以保存Integer值的内容。

要想写一个适用于包含Integer的list以及Integer的父类型，比如Integer，Number和Object，您可以指定List <？super Integer>。List<Integer>比List<? Super Integer>限制更大，因为前者只匹配一个Integer类型的list，而后者匹配任何Integer以及其子类型的list。

以下代码将数字1到10添加到列表的末尾：

public static void addNumbers(List<? super Integer> list) {

for (int i = 1; i <= 10; i++) {

list.add(i);

}

}

## 通配符和子类型

如泛型，继承和子类型所述，泛型类或接口不会仅仅因为它们之间类型存在关系而相关。但是，您可以使用通配符来创建泛型类或接口之间的关系。

给定以下两个常规（非泛型）类：

class A { /\* ... \*/ }

class B extends A { /\* ... \*/ }

编写以下代码是合理的：

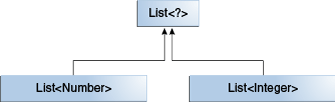
B b = new B();

A a = b;

此示例显示，常规类的继承遵循这个子类型规则：如果B继承A，则类B是类A的子类型。但是此规则不适用于泛型类型：

List<B> lb = new ArrayList<>();

List<A> la = lb; // compile-time error

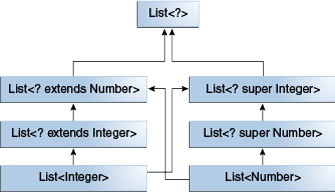
鉴于Integer是Number的子类型，List <Integer>和List <Number>之间的关系是什么？

虽然Integer是Number的子类型，但List <Integer>不是List <Number>的子类型，实际上这两种类型不相关。 List <Number>和List <Integer>的公共父项是List <？>。

为了创建这些类之间的关系，代码可以通过List <Integer>的元素访问Number的方法，使用上限通配符：

List<? extends Integer> intList = new ArrayList<>();

List<? extends Number> numList = intList; // OK. List<? extends Integer> is a subtype of List<? extends Number>

因为Integer是Number的子类型，而numList是Number对象的List，所以现在在intList（Integer对象List）和numList之间存在关系。下图显示了使用上边界和下边界通配符声明的几个List类之间的关系。

## 通配符捕获和帮助方法

在某些情况下，编译器推断通配符的类型。例如，一个list可以被定义为List <？>，但是当评估表达式时，编译器从代码中推断出特定类型。这种情况称为通配符捕获。

在大多数情况下，您不需要担心通配符捕获，除非您看到包含短语“捕获（capture of）”的错误消息。

WildcardError示例在编译时会产生捕获错误：

import java.util.List;

public class WildcardError {

void foo(List<?> i) {

i.set(0, i.get(0));

}

}

在本示例中，编译器将输入参数i处理为Object类型。当foo方法调用List.set（int，E）时，编译器无法确认正在插入到list中的对象的类型，并产生错误。当发生这种类型的错误时，通常意味着编译器认为您将错误的类型分配给变量。**之所以在Java语言中添加泛型是为了在编译时强制保证类型安全。**

WildcardError示例在使用Oracle JDK 7 javac实现编译时产生以下错误：

WildcardError.java:6: error: method set in interface List<E> cannot be applied to given types;

i.set(0, i.get(0));

^

required: int,CAP#1

found: int,Object

reason: actual argument Object cannot be converted to CAP#1 by method invocation conversion

where E is a type-variable:

E extends Object declared in interface List

where CAP#1 is a fresh type-variable:

CAP#1 extends Object from capture of ?

1 error

在这个例子中，代码正在尝试执行安全操作，那么你如何解决编译器错误？您可以通过编写私有的捕获通配符帮助方法来修复它。在这种情况下，您可以通过创建私有帮助方法fooHelper来解决问题，如WildcardFixed所示：

public class WildcardFixed {

void foo(List<?> i) {

fooHelper(i);

}

// Helper method created so that the wildcard can be captured

// through type inference.

private <T> void fooHelper(List<T> l) {

l.set(0, l.get(0));

}

}

由于帮助方法，编译器使用推断来确定T是调用中的CAP＃1（捕获变量）。该示例现在已成功编译。

按照惯例，帮助方法通常被命名为originalMethodNameHelper。

现在考虑一个更复杂的例子，WildcardErrorBad：

import java.util.List;

public class WildcardErrorBad {

void swapFirst(List<? extends Number> l1, List<? extends Number> l2) {

Number temp = l1.get(0);

l1.set(0, l2.get(0)); // expected a CAP#1 extends Number,

// got a CAP#2 extends Number;

// same bound, but different types

l2.set(0, temp); // expected a CAP#1 extends Number,

// got a Number

}

}

在这个例子中，代码正在尝试不安全的操作。例如，考虑以下调用swapFirst方法：

List<Integer> li = Arrays.asList(1, 2, 3);

List<Double> ld = Arrays.asList(10.10, 20.20, 30.30);

swapFirst(li, ld);

虽然List <Integer>和List <Double>都符合List<？extends Number>，从Integer的list中获取元素，并尝试将其放入Double值list中显然是不正确的。

使用Oracle的JDK javac编译器编译代码会产生以下错误：

WildcardErrorBad.java:7: error: method set in interface List<E> cannot be applied to given types;

l1.set(0, l2.get(0)); // expected a CAP#1 extends Number,

^

required: int,CAP#1

found: int,Number

reason: actual argument Number cannot be converted to CAP#1 by method invocation conversion

where E is a type-variable:

E extends Object declared in interface List

where CAP#1 is a fresh type-variable:

CAP#1 extends Number from capture of ? extends Number

WildcardErrorBad.java:10: error: method set in interface List<E> cannot be applied to given types;

l2.set(0, temp); // expected a CAP#1 extends Number,

^

required: int,CAP#1

found: int,Number

reason: actual argument Number cannot be converted to CAP#1 by method invocation conversion

where E is a type-variable:

E extends Object declared in interface List

where CAP#1 is a fresh type-variable:

CAP#1 extends Number from capture of ? extends Number

WildcardErrorBad.java:15: error: method set in interface List<E> cannot be applied to given types;

i.set(0, i.get(0));

^

required: int,CAP#1

found: int,Object

reason: actual argument Object cannot be converted to CAP#1 by method invocation conversion

where E is a type-variable:

E extends Object declared in interface List

where CAP#1 is a fresh type-variable:

CAP#1 extends Object from capture of ?

3 errors

对于这种问题没有帮助方法可以使用，因为代码设计从根本上就错了。

## 通配符使用指南

学习使用泛型进行编程时，更令人困惑的方面之一是确定何时使用上限通配符以及何时使用下限通配符。本页提供了设计代码时遵循的一些指导原则。

为了讨论的目的，将变量视为提供两个功能之一是有帮助的：

**一个”in”变量**

**”in”**变量为代码提供数据。想象一下有两个参数的复制方法：copy（src，dest）。 src参数提供要复制的数据，因此它是“in”参数。

**一个”out”变量**

“out”变量保存在其他地方使用的数据。在复制示例中，copy（src，dest），dest参数接受数据，所以它是“out”参数。

当然，一些变量同时用于“in”和“out”的目的 - 这种情况也在指南中解决。

在决定是否使用通配符以及通配符是什么类型时，可以使用“in”和“out”原则。以下列表提供了以下准则：

通配符指南：

* “in”变量用上限通配符定义，使用“extends”关键字
* “out”变量用下限通配符定义，使用“super”关键字
* 在“in”变量可以访问使用Object类中定义的方法情况下，使用无界通配符。
* 在代码需要访问变量作为“in”和“out”变量的情况下，请勿使用通配符。

这些准则不适用于方法的返回类型。应该避免使用通配符作为返回类型，因为它会强制程序员使用代码来处理通配符。

一个被定义为List<? Extends …>的list可以非正式地认为是只读，但这不是一个严格的保证。假设你有以下两个类：

class NaturalNumber {

private int i;

public NaturalNumber(int i) { this.i = i; }

// ...

}

class EvenNumber extends NaturalNumber {

public EvenNumber(int i) { super(i); }

// ...

}

请考虑以下代码：

List<EvenNumber> le = new ArrayList<>();

List<? extends NaturalNumber> ln = le;

ln.add(new NaturalNumber(35)); // compile-time error

因为List <EvenNumber>是List<? extends NaturalNumber>的子类型，可以将le分配给ln。但是您不能使用ln将natural number添加到even numbers的list中。但list中的以下操作是可能的：

* 你可以添加 null.
* 你可以调用clear方法.
* 你可以获取iterator并且调用它的remove方法.
* 您可以捕获通配符并写入从list中读取的元素。

你可以看到定义为List<? extends NaturalNumber>的list不是最严格意义上的只读，您可能会想到这样，因为您不能存储新元素或更改list中的现有元素。

# 类型擦除

泛型被引入Java语言，以便在编译时提供更严格的类型检查，并支持通用编程。为了实现泛型，Java编译器将类型擦除应用于：

* 在泛型类中用它的边界替换所有类型参数，或者以Object替代如果类型参数是无界的。因此，编译生成的字节码只包含普通的类，接口和方法。
* 必要时插入类型转换以保护类型安全。
* 生成桥接方法来保留扩展泛型类型中的多态。

类型擦除确保不会为参数化类型而创建新类;因此，泛型不会导致运行时额外的开销。

## 泛型类型的擦除

在类型擦除过程中，如果类型参数是有界的，则Java编译器将擦除所有类型参数，并将其替换为其边界第一个类型，如果类型参数为无边界，则该对象将替换为Object。

考虑以下通用类，表示单链表中的节点：

public class Node<T> {

private T data;

private Node<T> next;

public Node(T data, Node<T> next) }

this.data = data;

this.next = next;

}

public T getData() { return data; }

// ...

}

因为类型参数T是无界的，所以Java编译器用Object替换它：

public class Node {

private Object data;

private Node next;

public Node(Object data, Node next) {

this.data = data;

this.next = next;

}

public Object getData() { return data; }

// ...

}

在以下示例中，泛型Node类使用有界类型参数：

public class Node<T extends Comparable<T>> {

private T data;

private Node<T> next;

public Node(T data, Node<T> next) {

this.data = data;

this.next = next;

}

public T getData() { return data; }

// ...

}

Java编译器将有界类型参数T替换为第一个边界类，Comparable：

public class Node {

private Comparable data;

private Node next;

public Node(Comparable data, Node next) {

this.data = data;

this.next = next;

}

public Comparable getData() { return data; }

// ...

}

## 泛型方法的擦除

Java编译器还会擦除泛型方法参数中的类型参数。考虑以下泛型方法：

// Counts the number of occurrences of elem in anArray.

//

public static <T> int count(T[] anArray, T elem) {

int cnt = 0;

for (T e : anArray)

if (e.equals(elem))

++cnt;

return cnt;

}

因为T是无边界的，Java编译器用Object替换它：

public static int count(Object[] anArray, Object elem) {

int cnt = 0;

for (Object e : anArray)

if (e.equals(elem))

++cnt;

return cnt;

}

假设定义了以下类：

class Shape { /\* ... \*/ }

class Circle extends Shape { /\* ... \*/ }

class Rectangle extends Shape { /\* ... \*/ }

您可以编写一个泛型方法来绘制不同的形状：

public static <T extends Shape> void draw(T shape) { /\* ... \*/ }

Java编译器将T替换为

public static void draw(Shape shape) { /\* ... \*/ }

## 类型擦除和桥接方法的影响

有时类型擦除会导致您可能没有预料到的情况。以下示例显示了这种情况如何发生的。这个示例（在Bridge方法中描述）显示了编译器有时会创建一种称为桥接方法的合成方法，作为类型擦除过程的一部分。

给出以下两个类：

public class Node<T> {

public T data;

public Node(T data) { this.data = data; }

public void setData(T data) {

System.out.println("Node.setData");

this.data = data;

}

}

public class MyNode extends Node<Integer> {

public MyNode(Integer data) { super(data); }

public void setData(Integer data) {

System.out.println("MyNode.setData");

super.setData(data);

}

}

请考虑以下代码：

MyNode mn = new MyNode(5);

Node n = mn; // A raw type - compiler throws an unchecked warning

n.setData("Hello");

Integer x = mn.data; // Causes a ClassCastException to be thrown.

类型擦除后，此代码变为：

MyNode mn = new MyNode(5);

Node n = (MyNode)mn; // A raw type - compiler throws an unchecked warning

n.setData("Hello");

Integer x = (String)mn.data; // Causes a ClassCastException to be thrown.

下面是执行代码时会发生什么：

* n.setData("Hello");导致在MyNode类的对象上执行setData(Object)方法。（MyNode类从Node继承了setData(Object)。）
* 在setData（Object）的方法体中，由n引用的字段的对象被分配为String。
* 可以访问通过mn引用的同一对象的字段数据，并且预期它是一个整数（因为mn是一个MyNode，它是一个Node <Integer>。
* 尝试将一个String分配给Integer会导致Java编译器在分配时插入的转换的ClassCastException。

### 桥接方法

当编译一个类或接口，它是继承自参数化的类或接口时，编译器可能需要创建一个被称为桥接方法的合成方法，作为其类型擦除的一部分。您通常不需要担心桥接方法，但如果出现在堆栈跟踪中，则可能会感到困惑。

类型擦除后，Node和MyNode类变为：

public class Node {

public Object data;

public Node(Object data) { this.data = data; }

public void setData(Object data) {

System.out.println("Node.setData");

this.data = data;

}

}

public class MyNode extends Node {

public MyNode(Integer data) { super(data); }

public void setData(Integer data) {

System.out.println("MyNode.setData");

super.setData(data);

}

}

在经过类型擦除后，方法签名不匹配。 Node的方法变为setData（Object），而MyNode的方法为setData（Integer）。因此，MyNode setData方法不会覆盖Node 的setData方法。

为了解决这个问题，并在类型擦除后保留泛型类型的多态性，Java编译器会生成一个桥接方法，以确保子类型按预期工作。对于MyNode类，编译器为setData生成以下桥接方法：

class MyNode extends Node {

**// Bridge method generated by the compiler**

**//**

**public void setData(Object data) {**

**setData((Integer) data);**

**}**

public void setData(Integer data) {

System.out.println("MyNode.setData");

super.setData(data);

}

// ...

}

正如你所看到的，与Node类的setData方法有相同签名的桥接方法，在类型擦除之后将委托给原始的setData方法。

## 不可具体化类型

类型擦除部分讨论了编译器删除与type parameters 和type arguments相关的信息的过程。类型擦除有与变量参数（也称为可变参数）方法相关，其变量形式参数是不可具体化的类型。

### 不可具体化类型

可具体化类型是一种在整个运行时期它的类型信息都可用的类型。这包括原始类型，非泛型类型，原生类型，以及无边界通配符的调用。

不可具体化类型是其信息会在编译时期被类型擦除去除的类型那个。也就是指未定义为无边界通配符的泛型的调用。一个不可具体化类型它的所有信息在允许时期不是都可使用的。比如不可具体化类型List<String>和List<Number>，在运行时期JVM不能得知他们之间的类型差别。如“泛型的限制”章节所示，有种情况下，不能使用不可具体化类型类型：比如在instanceof表达式中，或者作为数组中的元素时。

### 堆污染

当参数化类型的变量指向的不是一个参数化类型的对象时，会发生堆污染。如果程序执行一些会在编译时产生“unchecked”警告的操作，则会出现这种情况。如果在编译时（在编译时类型检查规则的限制内）或在运行时，如果涉及参数化类型（例如，转换或方法调用）的操作的正确性不可得到验证的话，就会产生一个”unchecked”警告。例如，当混合原生类型和参数化类型时，或者执行未检查的转换时，会发生堆污染。

在正常情况下，当同时编译所有代码时，编译器会发出一个”unchecked”警告，以引起您对潜在的堆污染的注意。如果您单独编译部分代码，则很难发现堆污染的潜在风险。如果确保您的代码编译没有警告，则不会有堆污染的发生。

### 使用不可具体化形式参数的可变参数方法的潜在漏洞

包含可变输入参数的泛型方法可能导致堆污染。

考虑下面的ArrayBuilder类。

public class ArrayBuilder {

public static <T> void addToList (List<T> listArg, T... elements) {

for (T x : elements) {

listArg.add(x);

}

}

public static void faultyMethod(List<String>... l) {

Object[] objectArray = l; // Valid

objectArray[0] = Arrays.asList(42);

String s = l[0].get(0); // ClassCastException thrown here

}

}

下面示例中，HeapPollutionExample 使用了 ArrayBuiler 类:

public class HeapPollutionExample {

public static void main(String[] args) {

List<String> stringListA = new ArrayList<String>();

List<String> stringListB = new ArrayList<String>();

ArrayBuilder.addToList(stringListA, "Seven", "Eight", "Nine");

ArrayBuilder.addToList(stringListB, "Ten", "Eleven", "Twelve");

List<List<String>> listOfStringLists =

new ArrayList<List<String>>();

ArrayBuilder.addToList(listOfStringLists,

stringListA, stringListB);

ArrayBuilder.faultyMethod(Arrays.asList("Hello!"), Arrays.asList("World!"));

}

}

编译时，ArrayBuilder.addToList方法的定义会产生以下警告：

warning: [varargs] Possible heap pollution from parameterized vararg type T

当编译器遇到一个可变参数方法，它将可变形式参数转换到一个数组中。然而Java编程语言不允许创建参数化类型的数组。在方法arrayBuilder.addToList中，编译器将可变形式参数T…elements转换成一个形式参数T[] elements数组。然而由于类型擦除，编译器将可变形式参数转换为Object[] elements。因此，这儿有堆污的可能性。

以下语句将可变形式参数l分配给Object数组objectArgs：

Object[] objectArray = l;

这个声明可能会引起堆污。一个与可变形式参数l的参数化类型不相匹配的值可以被分配给变量objectArray，因此可以分配给l。但是，编译器在此语句中不会生成”unchecked“警告。当编译器将形式参数List <String> ... l转换为形式参数List [] l时，编译器已经生成了一个警告。因此这个声明是有效的;这里变量l是List []类型，而List []是Object []的子类型。

因此，如果将任何类型的List对象分配给objectArray数组的任何部分，编译器不会发出警告或错误，就如下面语句所展示：

objectArray[0] = Arrays.asList(42);

这个语句将一个包含一个Integer对象的List对象分配给了objectArray数组的第一个索引

假如你使用以下语句调用ArrayBuilder.faultyMethod

ArrayBuilder.faultyMethod(Arrays.asList("Hello!"), Arrays.asList("World!"));

在运行时期，以下语句JVM抛出一个ClassCastException异常

// ClassCastException thrown here

String s = l[0].get(0);

存储在变量l的第一个索引处的对象是List <Integer>类型，但此语句却期待返回一个List <String>类型的对象。

### 阻止来自使用不可具体化形式参数的可变参数方法的警告

如果你声明一个可变参数方法有一个参数化类型的参数，并且你确保方法体内不会因不正确处理可变形式参数而抛出一个ClassCastException或者其他类似的异常，你可以通过向静态和非构造函数方法声明中添加以下注释，来防止编译器为这些类型的可变参数方法生成的警告：

@SafeVarargs

@SafeVarargs注释是方法合同的一个文档部分; 此注释声明该方法的实现将不会不正确地处理可变形式参数。

尽管不太可取，但也可以通过在方法声明中加入以下内容来抑制这种警告：

@SuppressWarnings({"unchecked", "varargs"})

但是，这种方法不会抑制从该方法调用处产生的警告。如果您不熟悉@SuppressWarnings语法，请参阅”注释“。

# 泛型的限制

要有效地使用Java泛型，您必须考虑以下限制：

* 无法使用原始类型实例化泛型类型
* 无法创建类型参数实例
* 无法声明类型参数静态字段
* 无法在参数化类型上使用类型转换或instranceof
* 无法创建参数化类型数组
* 无法创建、捕获、或抛出参数化类型对象
* 无法重载一个形式参数被擦除后是相同原生类型的方法

## 无法使用原始类型实例化泛型类型

考虑以下参数化类型

lass Pair<K, V> {

private K key;

private V value;

public Pair(K key, V value) {

this.key = key;

this.value = value;

}

// ...

}

在创建一个Pair对象时，你不可以用原始类型来替代类型参数K或者V

Pair<**int, char**> p = new Pair<>(8, 'a'); // compile-time error

你只可以使用非原始类型来替代类型参数K和V

Pair<**Integer, Character**> p = new Pair<>(8, 'a');

请注意，Java编译器将8自动装箱为Integer.valueOf（8）并且“a”自动装箱为Character（'a'）：

Pair<Integer, Character> p = new Pair<>(Integer.valueOf(8), new Character('a'));

## 无法创建类型参数实例

您不能创建一个类型参数的实例。例如，以下代码会导致编译时错误：

public static <E> void append(List<E> list) {

E elem = new E(); // compile-time error

list.add(elem);

}

作为解决方法，您可以通过反射创建类型参数的对象：

public static <E> void append(List<E> list, Class<E> cls) throws Exception {

E elem = cls.newInstance(); // OK

list.add(elem);

}

您可以如下调用append方法：

List<String> ls = new ArrayList<>();

append(ls, String.class);

## 无法声明类型参数静态字段

一个类的静态字段是由类的所有非静态对象共享的类级变量。因此，不允许使用类型参数的静态字段。考虑下列类：

public class MobileDevice<T> {

private static T os;

// ...

}

如果允许类型参数的静态字段，那么以下代码将被混淆：

MobileDevice<Smartphone> phone = new MobileDevice<>();

MobileDevice<Pager> pager = new MobileDevice<>();

MobileDevice<TabletPC> pc = new MobileDevice<>();

因为静态字段os是由phone，pager和pc共享的，那么os的实际类型是什么？它不能同时是Smartphone，Pager和TabletPC。因此，您不能创建类型参数的静态字段。因此，您不能创建类型参数的静态字段。

## 无法在参数化类型上使用类型转换或instranceof

因为Java编译器会擦除通用代码中的所有类型参数，所以无法验证在运行时使用泛型类型的参数化类型：

public static <E> void rtti(List<E> list) {

if (list instanceof ArrayList<Integer>) { // compile-time error

// ...

}

}

传递给rtti方法的一系列参数化类型是：

S = { ArrayList<Integer>, ArrayList<String> LinkedList<Character>, ... }

运行时不跟踪类型参数，因此它不能区分ArrayList <Integer>和ArrayList <String>之间的区别。您可以做的最多的是使用无界通配符来验证list是否为ArrayList：

public static void rtti(List<?> list) {

if (list instanceof ArrayList<?>) { // OK; instanceof requires a reifiable type

// ...

}

}

通常，您不能转换为参数化类型，除非由无界通配符来参数化的。例如：

List<Integer> li = new ArrayList<>();

List<Number> ln = (List<Number>) li; // compile-time error

然而，在某些情况下，编译器知道一个类型参数始终是有效的并允许该转换。例如：

List<String> l1 = ...;

ArrayList<String> l2 = (ArrayList<String>)l1; // OK

## 无法创建参数化类型数组

您不能创建参数化类型的数组。例如，以下代码无法编译：

List<Integer>[] arrayOfLists = new List<Integer>[2]; // compile-time error

以下代码说明了将不同类型插入到数组中时会发生什么：

Object[] strings = new String[2];

strings[0] = "hi"; // OK

strings[1] = 100; // An ArrayStoreException is thrown.

如果您使用泛型列表尝试相同的事情，则会出现问题：

Object[] stringLists = new List<String>[]; // compiler error, but pretend it's allowed

stringLists[0] = new ArrayList<String>(); // OK

stringLists[1] = new ArrayList<Integer>(); // An ArrayStoreException should be thrown,

// but the runtime can't detect it.

如果允许使用参数化列表的数组，以前的代码将无法抛出所需的ArrayStoreException。

## 无法创建、捕获、或抛出参数化类型对象

泛型类不能直接或间接继承Throwable类。例如，以下类将不会编译：

// Extends Throwable indirectly

class MathException<T> extends Exception { /\* ... \*/ } // compile-time error

// Extends Throwable directly

class QueueFullException<T> extends Throwable { /\* ... \*/ // compile-time error

一个方法不能捕获一个类型参数的实例：

public static <T extends Exception, J> void execute(List<J> jobs) {

try {

for (J job : jobs)

// ...

} catch (T e) { // compile-time error

// ...

}

}

但是，您可以在throws子句中使用类型参数：

class Parser<T extends Exception> {

public void parse(File file) throws T { // OK

// ...

}

}

## 无法重载一个形式参数被擦除后是相同原生类型的方法

一个类不能有两个在类型擦除后将具有相同的签名重载方法。

public class Example {

public void print(Set<String> strSet) { }

public void print(Set<Integer> intSet) { }

}

重载将共享相同的类文件表示，并将生成编译时错误。